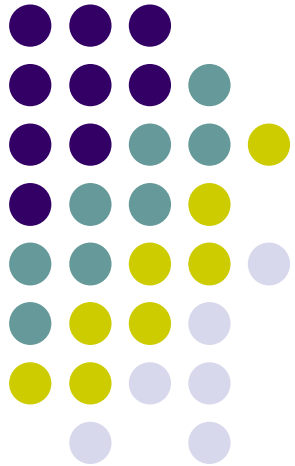
Mata Kuliah : Sistem Operasi

Kode MK : IT-012336

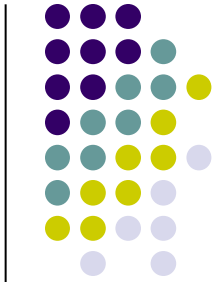
7

Sinkronisasi

Tim Teaching Grant
Mata Kuliah Sistem Operasi



Proses Sinkronisasi

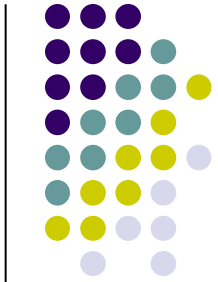


- Latar Belakang
- Masalah Critical Section
- Sinkronisasi Hardware
- Semaphores
- Monitors

Overview (1)

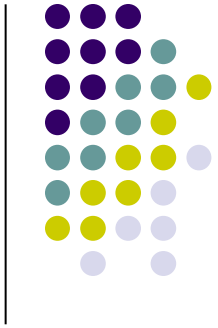
- *Proteksi OS:*
  - *Independent* process tidak terpengaruh atau dapat mempengaruhi eksekusi/data proses lain.
- “Concurrent Process”
  - OS: mampu membuat banyak proses pada satu saat
  - Proses-proses bekerja-sama: sharing data, pembagian task, passing informasi dll
  - Proses => mempengaruhi proses lain dalam menggunakan data/informasi yang sengaja di-”share”
- *Cooperating* process – sekumpulan proses yang dirancang untuk saling bekerja-sama untuk mengerjakan task tertentu.

Overview (2)



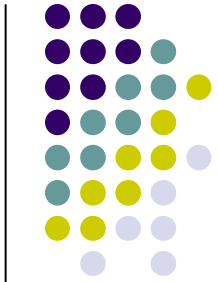
- Keuntungan kerja-sama antar proses
  - Information sharing: file, DB => digunakan bersama
  - Computation speed-up: parallel proses
  - Modularity: aplikasi besar => dipartisi dalam banyak proses.
  - Convenience: kumpulan proses => tipikal lingkungan kerja.
- “Cooperating Process”
  - Bagaimana koordinasi antar proses? Akses/Update data
  - Tujuan program/task: integritas, **konsistensi** data dapat dijamin

Latar Belakang



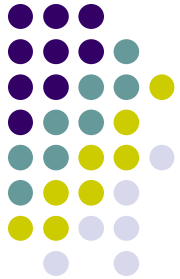
- Menjamin konsistensi data:
  - Program/task-task dapat menghasilkan operasi yang benar setiap waktu
  - Deterministik: untuk input yang sama hasil harus sama (sesuai dengan logika/algroitma program).
- Contoh: Producer – Consumer
  - Dua proses: producer => menghasilkan informasi; consumer => menggunakan informasi
  - Sharing informasi: buffer => tempat penyimpanan data
    - *unbounded-buffer,* penempatan tidak pada limit praktis dari ukuran buffer
    - *bounded-buffer* diasmumsikan terdapat ukuran buffer yang tetap

Bounded Buffer (1)



- Implementasi buffer:
  - IPC: komunikasi antar proses melalui messages membaca/menulis buffer
  - Shared memory: programmer secara eksplisit melakukan “deklarasi” data yang dapat diakses secara bersama.
  - Buffer dengan ukuran n => mampu menampung n data
    - Producer mengisi data buffer => increment “counter” (jumlah data)
    - Consumer mengambil data buffer => decrement “counter”
    - Buffer, “counter” => shared data (update oleh 2 proses)

Bounded Buffer (2)



- Producer process

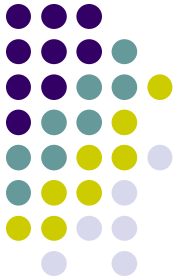
```
repeat      …
    produce an item in nextp
    …
    while counter = n do no-op;
    buffer [in] := nextp;
    in := in + 1 mod n;
    counter := counter +1;

until false;
```

- Consumer process

```
repeat
    while counter = 0 do no-op;
    nextc := buffer [out];
    out := out + 1 mod n;
    counter := counter – 1;
    …
    consume the item in nextc
    …
until false;
```

Bounded Buffer (4)



- Apakah terdapat jaminan operasi akan benar jika berjalan concurrent?
- Misalkan: counter = 5
  - Producer: counter = counter + 1;
  - Consumer: counter = counter - 1;
  - Nilai akhir dari counter?
- Operasi concurrent P & C =>
  - Operasi dari high level language => sekumpulan instruksi mesin: “increment counter”

    Load Reg1, Counter

    Add Reg1, 1

    Store Counter, Reg1

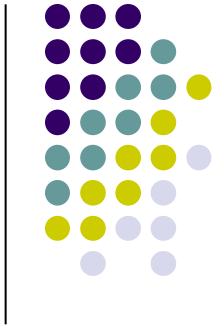
Bounded Buffer (5)

- “decrement counter”

  Load Reg2, Counter
  Subtract Reg2, 1
  Store Counter, Reg2

- Eksekusi P & C tergantung scheduler (dapat gantian)
  - **T0: Producer : Load Reg1, Counter (Reg1 = 5)**
  - **T1: Producer : Add Reg1, 1 (Reg1 = 6)**
  - **T2: Consumer: Loag Reg2, Counter (Reg2 = 5)**
  - **T3: Consumer: Subtract Reg1, 1 (Reg2 = 4)**
  - **T4: Producer: Store Counter, Reg1 (Counter = 6)**
  - **T5: Consumer: Store Counter, Reg2 (Counter = 4)**

Race Condition

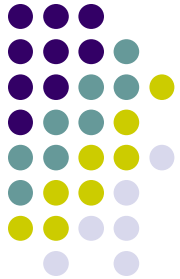


- Concurrent C & P
  - Shared data “counter” dapat berakhir dengan nilai: 4, atau 5, atau 6
  - Hasilnya dapat salah dan tidak konsisten
- Race Condition:
  - Keadaan dimana lebih dari satu proses meng-update data secara “concurrent” dan hasilnya sangat bergantung dari urutan proses mendapat jatah CPU (run)
  - Hasilnya tidak menentu dan tidak selalu benar
  - Mencegah race condition: sinkronisasi proses dalam meng-update shared data

Sinkronisasi

- Sinkronisasi:
  - Koordinasi akses ke shared data, misalkan hanya satu proses yang dapat menggunakah shared var.
  - Contoh operasi terhadap var. “counter” harus dijamin di-eksekusi dalam satu kesatuan (atomik) :
    - *counter* := *counter* + 1;
    - *counter* := *counter* - 1;
  - Sinkronisasi merupakan “issue” penting dalam rancangan/implementasi OS (shared resources, data, dan multitasking).

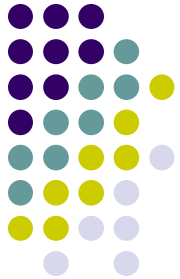
Masalah Critical Section



- n proses mencoba menggunakan shared data bersamaan
- Setiap proses mempunyai “code” yang mengakses/ manipulasi shared data tersebut => “critical section”
- Problem: Menjamin jika ada satu proses yang sedang “eksekusi” pada bagian “critical section” tidak ada proses lain yang diperbolehkan masuk ke “code” critical section dari proses tersebut.
- Structure of process *Pi*

```
repeat
    entry section
        critical section
    exit section
        reminder section
until false;
```

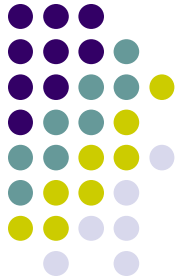
Solusi Masalah Critical Section



- Ide :
  - Mencakup pemakaian secara “exclusive” dari shared variable tersebut
  - Menjamin proses lain dapat menggunakan shared variable tersebut
- Solusi “critical section problem” harus memenuhi:
  1. **Mutual Exclusion:** Jika proses Pi sedang “eksekusi” pada bagian “critical section” (dari proses Pi) maka tidak ada proses proses lain dapat “eksekusi” pada bagian critical section dari proses-proses tersebut.

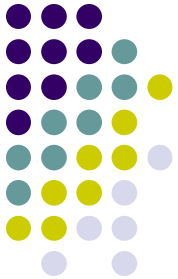
  2. **Progress:** Jika tidak ada proses sedang eksekusi pada critical section-nya dan jika terdapat lebih dari satu proses lain yang ingin masuk ke critical section, maka pemilihan siapa yang berhak masuk ke critical section tidak dapat ditunda tanpa terbatas.

Solusi (cont.)



3. **Bounded Waiting**: Terdapat batasan berapa lama suatu proses harus menunggu giliran untuk mengakses “critical section” – jika seandainya proses lain yang diberikan hak akses ke critical section.
   - Menjamin proses dapat mengakses ke “critical section” (tidak mengalami starvation: proses se-olah berhenti menunggu request akses ke critical section diperbolehkan).
   - Tidak ada asumsi mengenai kecepatan eksekusi proses proses n tersebut.

Solusi Sederhana : Kasus 2 proses

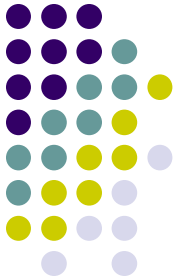


- Hanya 2 proses
- Struktur umum dari program code ***Pi*** dan ***Pj***:

```
repeat
        entry section
                critical section
        exit section
                reminder section
until false;
```

- Software solution: merancang algoritma program untuk solusi critical section
  - Proses dapat mengunakan “common var.” untuk menyusun algoritma tsb.

Algoritma 1



- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn - i** $\Rightarrow$ P_i dapat masuk ke criticalsection
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

- Mutual exclusion terpenuhi, tetapi menentang progress

Algoritma 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i siap dimasukkan ke dalam critical section
- Process P_i

```
do {
    flag[i] := true;
    while (flag[j]) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Mutual exclusion terpenuhi tetapi progress belum terpenuhi.

Algoritma 3

- Kombinasi shared variables dari algoritma 1 and 2.
- Process P_i

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Ketiga kebutuhan terpenuhi, solusi masalah critical section pada dua proses

Algoritma Bakery

Critical section untuk n proses

- Sebelum proses akan masuk ke dalam “critical section”, maka proses harus mendapatkan “nomor” (tiket).
- Proses dengan nomor terkecil berhak masuk ke critical section.
  - Jika proses P_i dan P_j menerima nomor yang sama, jika $i < j$, maka P_i dilayani pertama; jika tidak P_j dilayani pertama
- Skema penomoran selalu dibuat secara berurutan, misalnya 1,2,3,3,3,3,4,5...

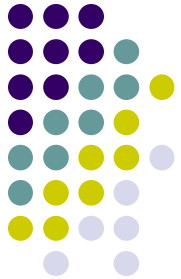
Algoritma Bakery (2)

- Notasi <≡ urutan lexicographical (ticket #, process id #)
  - (*a,b*) < *c,d*) jika *a* < *c* atau jika *a* = *c* and *b* < *d*
  - max (a_0,…, a_{n-1}) dimana a adalah nomor, *k*, seperti pada $k \geq a_i$ untuk *i* - 0, …, *n* – 1
- Shared data

  var *choosing*: **array** [0..*n* – 1] **of** *boolean*

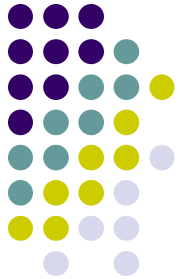
  number: **array** [0..*n* – 1] **of** *integer*,
- Initialized: choosing =: *false* ; number => 0

Algoritma Bakery (3)



```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n –
    1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

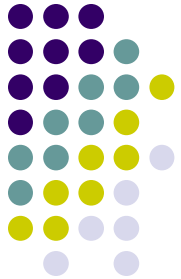
Sinkronisasi Hardware



- Memerlukan dukungan hardware (prosesor)
  - Dalam bentuk “instruction set” khusus: test-and-set
  - Menjamin operasi atomik (satu kesatuan): test nilai dan ubah nilai tersebu
- Test-and-Set dapat dianalogikan dengan kode:

```
function Test-and-Set (var target:boolean): boolean;
        begin
                Test-and-Set := target;
                target := true;
        end;
```

Test-and-Set (mutual exclusion)



- Mutual exclusion dapat diterapkan:
  - Gunakan shared data,

    variabel: *lock: boolean (initially false)*
  - lock: menjaga critical section
- Process Pi:

```
do {
    while (TestAndSet(lock)) ;
   critical section
   lock = false;
   remainder section
   }
```

Semaphore

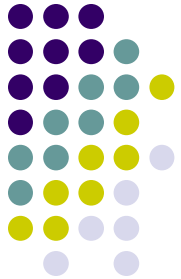
- Perangkat sinkronisasi yang tidak membutuhkan *busy waiting*
- Semaphore S – integer variable
  - Dapat dijamin akses ke var. S oleh dua operasi atomik:
    - wait (S): while S ≤ 0 do no-op;
      S := S – 1;
    - signal (S): S := S + 1;

Contoh : n proses

- Shared variables
  - var mutex : semaphore
  - initially mutex = 1
- Process Pi

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

Implementasi Semaphore

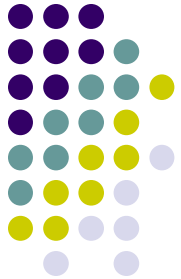


- Didefinisikan sebuah Semaphore dengan sebuah record

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Diasumsikan terdapat 2 operasi sederhana :
  - **block** menhambat proses yang akan masuk
  - **wakeup(*P*)** memulai eksekusi pada proses P yang di block

Implementasi Semaphore (2)



- Operasi Semaphore-nya menjadi :

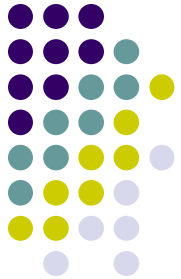
wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(*S*):

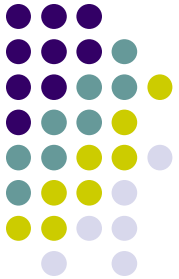
```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```

Masalah Klasik Sinkronisasi



- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem



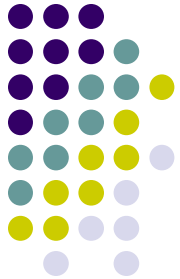
- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem : Producer-Consumer



Producer:

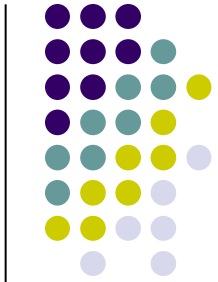
```
do
  .. produce item pada nextp;
    wait(empty_slot);
    wait(mutex);
...(critical section)
    … add nextp to buffer;
    …
    signal(mutex);
    signal(full_item);

while (true);
```

Consumer:

```
do
    wait(full_item);
    wait(mutex);
...(critical section)
    … remove nextp buffer;
    …
    signal(mutex);
    signal(empty_slot);
    .. consume item nextp;
while (true);
```

Readers-Writers Problem



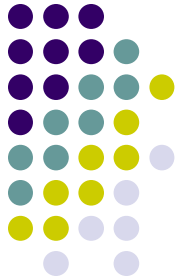
- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem (2)



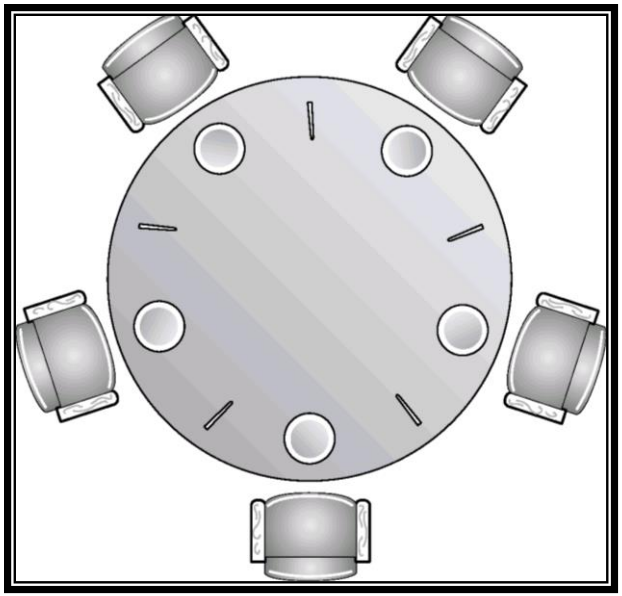
- Writters Process

```
wait(wrt);
                    …
    writing is performed
                    …
signal(wrt);
```

- Readers Process

```
wait(mutex);
readcount++;
if (readcount == 1)
     wait(rt);
signal(mutex);
      …
     reading is performed
      …
wait(mutex);
readcount--;
if (readcount == 0)
     signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem

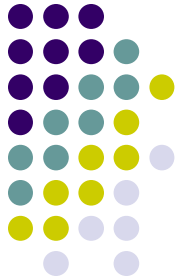


- Shared data

  semaphore chopstick[5];

Semua inisialisasi bernilai 1

Dining-Philosophers Problem



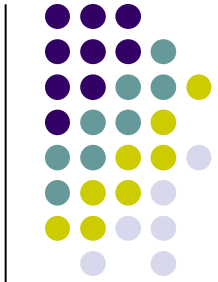
- Philosopher *i*:

```
do {
   wait(chopstick[i])
   wait(chopstick[(i+1) % 5])
      …
      eat
      …
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      …
      think
      …
} while (1);
```

Solusi Tingkat Tinggi

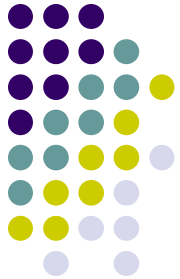
- **Motif:**
  - Operasi wait(S) dan signal(S) tersebar pada code program => manipulasi langsung struktur data semaphore
  - Bagaimana jika terdapat bantuan dari lingkungan HLL (programming) untuk sinkronisasi ?
  - Pemrograman tingkat tinggi disediakan sintaks-sintaks khusus untuk menjamin sinkronisasi antar proses, thread
- **Misalnya:**
  - Monitor & Condition
  - Conditional Critical Region

Monitor



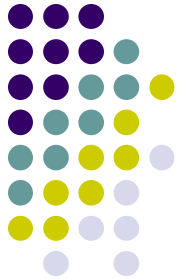
- **Monitor mensinkronisasi sejumlah proses:**
  - suatu saat hanya satu yang aktif dalam monitor dan yang lain menunggu
- **Bagian dari bahasa program (mis. Java).**
  - Tugas compiler menjamin hal tersebut terjadi dengan menerjemahkan ke “low level synchronization” (semphore, instruction set dll)
- Cukup dengan statement (deklarasi) suatu section/fungsi adalah monitor => mengharuskan hanya ada satu proses yang berada dalam monitor (section) tsb

Monitor (2)



```
type monitor-name = monitor
        variable declarations
        procedure entry P1 :(…);
                begin … end;
        procedure entry P2(…);
                begin … end;
                        :
        procedure entry Pn (…);
                begin…end;
        begin
```

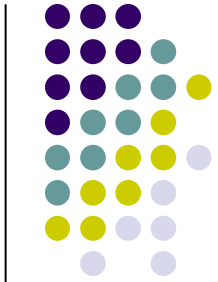
Monitor (3)



- Proses-proses harus disinkronisasikan di dalam monitor:
  - Memenuhi solusi critical section.
  - Proses dapat menunggu di dalam monitor.
  - Mekanisme: terdapat variabel (condition) dimana proses dapat menguji/menunggu sebelum mengakses “critical section”

var *x, y: condition*

Monitor (4)



- **Condition:** memudahkan programmer untuk menulis code pada monitor**.**

  Misalkan : var x: condition ;

- Variabel condition hanya dapat dimanipulasi dengan operasi: **wait() dan signal()**
  - x.wait() jika dipanggil oleh suatu proses maka proses tsb. akan suspend - sampai ada proses lain yang memanggil: x. signal()
  - x.signal() hanya akan menjalankan (resume) 1 proses saja yang sedang menunggu (suspend) (tidak ada proses lain yang wait maka tidak berdampak apapun)

Skema Monitor